15 | 查询商品：资源不足有哪些性能表现？

2021-04-23 高楼

高楼的性能工程实战课 进入课程 >

讲述：高楼

时长 26:48 大小 24.55M

你好，我是高楼。

这节课，我们来收拾“查询商品”这个接口。虽然这次的现象同样是 TPS 低、响应时间长，但是，这个接口走的路径和之前的不一样，所以在分析过程中会有些新鲜的东西，你将看到在资源真的不足的情况下，我们只有增加相应节点的资源才能提升性能。

在我的项目中，我一直都在讲，**不要轻易给出资源不足的结论。因为但凡有优化的空间，我们都要尝试做优化，而不是直接告诉客户加资源。而给出“增加资源”这个结论，也必须建立在有足够证据的基础上。**在这节课中，你也将看到这一点。

话不多说，我们直接开始今天的内容。

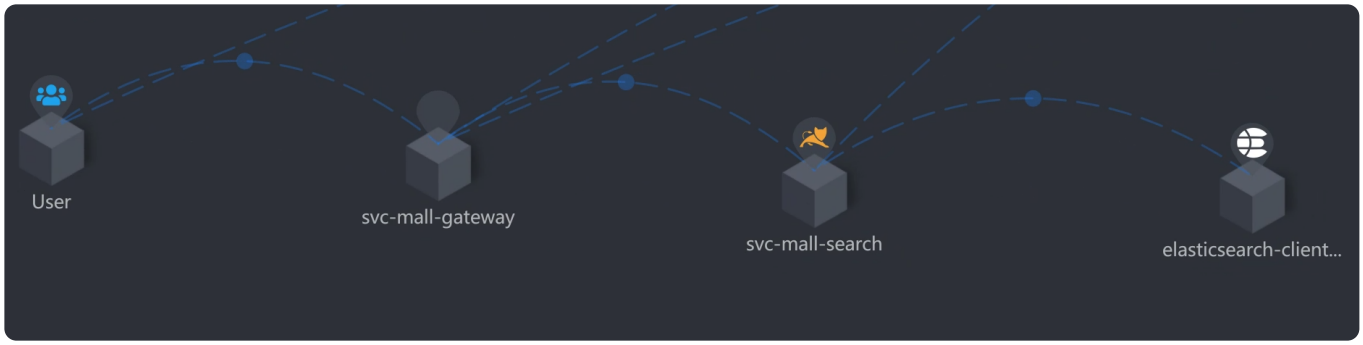
压力场景数据

对于查询商品接口，我们第一次试执行性能场景的结果如下：

你看，TPS 只有 250 左右，并且响应时间也明显随着压力的增加而增加了，看起来瓶颈已经出现了，对吧？根据哥的逻辑，下一步就是看架构图啦。

先看架构图

我们用 APM 工具来看看这个接口的架构。



你看，从压力机到 Gateway 服务、到 Search 服务、再到 ES-Client，这个 APM 工具也只能帮我们到这里了。因为我们用的是 ElasticSearch 7 来做的搜索服务的支撑，而这个 skywalking 工具也没有对应的 Agent，所以后面并没有配置 skywalking。

在这里，我要多啰嗦几句。现在的 APM 工具大多是基于应用层来做的，有些运维 APM 采集的数据会更多一些，但也是响应时间、吞吐量等这样的信息。对于性能分析而言，现在的 APM 工具有减少排查时间的能力，但是在组件级的细化定位上还有待提高。虽然 AI OPS 也被提到了台面，但是也没见过哪个公司上了 AIOPS 产品后，就敢不让人看着。

总之，从细化分析的角度，我们在定位问题的根本原因时，手头有什么工具就可以用什么工具，即使什么工具都没有，撸日志也是照样能做到的，所以我建议你不要迷信工具，要“迷信”思路。

下面我们来拆分下这个接口的响应时间，看看这个案例的问题点在哪里。

拆分响应时间

“在 RESAR 性能分析逻辑中，拆分响应时间只是一个分析的起点。”这是我一直在强调的一句话。如果性能工程师连这个都不会做，就只能好好学习天天向上了。

根据架构图，我们拆分响应时间如下。

Gateway 服务上的响应时间：

Search 服务上的响应时间：

ES Client 的响应时间：

一层层看过之后，我们发现查询商品这个接口的响应时间消耗在了 ES client 上面。而在我们这个查询的路径上，在 gateway/search 服务上，我们并没有做什么复杂的动作。

既然知道了响应时间消耗在哪里，下面我们就来定位它，看能不能把 TPS 优化起来。

全局监控

根据高老师的经验，我们还是从全局监控开始，看全局监控可以让我们更加有的放矢。在分析的过程中，经常有人往下走了几步之后，就开始思维混乱、步伐飘逸。因为数据有很多，所以分析时很容易从一个数据走到不重要的分支上去了。而这时候，如果你心里有全局监控数据，思路就会更清晰，不会在无关的分支上消耗时间。

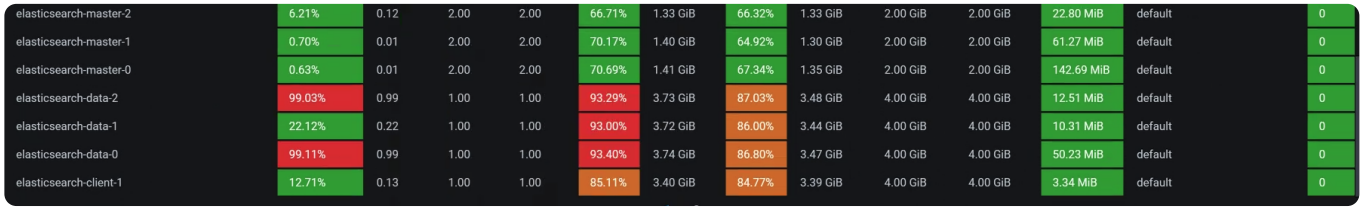
回到我们这个例子中，从下面的 k8s worker（也就是 k8s 中的 node，在我们的环境中我习惯叫成 worker，就是为了体现：在我的地盘，我爱叫啥就叫啥）的数据上来看，似乎没有一个 worker 的资源使用率是特别高的。

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.00 week	7.64 GiB	4	0.56	9.67%	24.09%	29.95%	0 B/s	84.01 kB/s	294	34	401.39 kbps	467.71 kbps
k8s-master-2	k8s-master-2	3.59 week	7.64 GiB	4	1.13	20.13%	23.08%	29.95%	0 B/s	76.15 kB/s	281	32	248.39 kbps	278.36 kbps
k8s-master-3	k8s-master-3	3.59 week	7.64 GiB	4	0.30	8.02%	24.40%	29.95%	0 B/s	78.34 kB/s	321	43	984.23 kbps	984.23 kbps
k8s-worker-1	k8s-worker-1	10.57 hour	15.51 GiB	6	0.10	5.92%	51.29%	44.69%	13.81 kB/s	23.00 kB/s	29	93	7.87 Mbps	8.07 Mbps
k8s-worker-2	k8s-worker-2	10.56 hour	15.51 GiB	6	0.87	15.34%	22.22%	29.95%	18.42 kB/s	97.18 kB/s	29	18 K	15.27 Mbps	14.92 Mbps
k8s-worker-3	k8s-worker-3	10.55 hour	15.51 GiB	6	1.29	17.79%	39.25%	32.96%	27.62 kB/s	15.36 kB/s	32	502	13.93 Mbps	13.81 Mbps
k8s-worker-4	k8s-worker-4	2.50 week	15.51 GiB	6	0.45	9.48%	36.78%	31.48%	9.21 kB/s	9.98 kB/s	47	114	4.80 Mbps	5.67 Mbps
k8s-worker-5	k8s-worker-5	2.50 week	15.51 GiB	6	7.20	54.64%	65.36%	36.51%	41.43 kB/s	21.89 kB/s	28	119	9.36 Mbps	18.43 Mbps
k8s-worker-6	k8s-worker-6	2.50 week	15.51 GiB	6	4.30	53.35%	64.95%	39.10%	18.41 kB/s	6.88 kB/s	29	111	13.02 Mbps	17.27 Mbps
k8s-worker-7	k8s-worker-7	2.20 week	15.51 GiB	4	0.92	13.42%	37.48%	40.70%	15.92 kB/s	4.46 kB/s	28	116	3.92 Mbps	6.81 Mbps
k8s-worker-8	k8s-worker-8	1.02 week	15.51 GiB	4	0.25	10.51%	22.03%	44.23%	0 B/s	3.64 kB/s	29	321	1.55 Mbps	370.20 kbps
k8s-worker-9	k8s-worker-9	2.20 week	15.51 GiB	6	0.17	4.72%	26.33%	29.95%	4.60 kB/s	2.48 kB/s	28	81	507.39 kbps	918.62 kbps

请你注意，在 k8s 中看资源消耗，一定不要只看 worker 这个层面，因为这个层面还不够，一个 worker 上可能会运行多个 pod。从上图来看，由于 worker 层面没有资源消耗，但是时间又明显是消耗在 ES client 上的，所以，接下来我们要看一下每一个 pod 的资源使用情况。

Pod名称	CPU%(总核)	使用核数	CPU需求	CPU限制	WSS%	WSS	RSS%	RSS	内存需求	内存限制	磁盘	命名空间	重启
node-exporter-dmbg6	4.47%	0.01	0.10	0.25	6.31%	11.36 MiB	5.43%	9.77 MiB	180.00 MiB	180.00 MiB	20.00 KiB	monitoring	0
node-exporter-rt2nr	2.20%	0.00	0.01	0.02	18.36%	7.34 MiB	18.36%	7.34 MiB	20.00 MiB	40.00 MiB	16.00 KiB	monitoring	0
node-exporter-hdlhd	2.15%	0.00	0.01	0.02	17.78%	7.11 MiB	17.78%	7.11 MiB	20.00 MiB	40.00 MiB	16.00 KiB	monitoring	0
elasticsearch-data-1	15.88%	0.16	1.00	1.00	92.98%	3.72 GiB	86.00%	3.44 GiB	4.00 GiB	4.00 GiB	10.31 MiB	default	0
grafana-57877c9c47-cmt98	21.19%	0.04	0.10	0.20	23.97%	47.95 MiB	18.14%	36.29 MiB	100.00 MiB	200.00 MiB	576.00 KiB	monitoring	0
alertmanager-main-1	0.03%	0.00	0.10	0.10	22.98%	5.75 MiB	22.98%	5.75 MiB	25.00 MiB	25.00 MiB	16.00 KiB	monitoring	0
elasticsearch-master-2	8.08%	0.16	2.00	2.00	66.70%	1.33 GiB	66.32%	1.33 GiB	2.00 GiB	2.00 GiB	23.25 MiB	default	0
node-exporter-wcsj7	1.48%	0.00	0.10	0.25	7.20%	12.96 MiB	7.20%	12.96 MiB	180.00 MiB	180.00 MiB	20.00 KiB	monitoring	1
elasticsearch-client-0	74.74%	0.75	1.00	1.00	85.59%	3.42 GiB	85.16%	3.41 GiB	4.00 GiB	4.00 GiB	64.95 MiB	default	0
node-exporter-r5m4b	2.06%	0.01	0.10	0.25	7.01%	12.62 MiB	7.01%	12.62 MiB	180.00 MiB	180.00 MiB	20.00 KiB	monitoring	0
elasticsearch-data-0	99.70%	1.00	1.00	1.00	93.40%	3.74 GiB	86.80%	3.47 GiB	4.00 GiB	4.00 GiB	50.23 MiB	default	0
node-exporter-xqzqf	1.61%	0.00	0.10	0.25	6.64%	11.95 MiB	6.64%	11.95 MiB	180.00 MiB	180.00 MiB	20.00 KiB	monitoring	1
node-exporter-7v467	1.22%	0.00	0.10	0.25	7.05%	12.68 MiB	7.04%	12.68 MiB	180.00 MiB	180.00 MiB	20.00 KiB	monitoring	1
jenkins-574c5c95b-rjzfb	0.08%	0.00	2.00	2.00	36.59%	749.41 MiB	34.69%	710.52 MiB	2.00 GiB	2.00 GiB	3.43 MiB	default	0
node-exporter-r5m4b	2.18%	0.00	0.01	0.02	26.78%	10.71 MiB	26.78%	10.71 MiB	20.00 MiB	40.00 MiB	16.00 KiB	monitoring	0
node-exporter-q79lz	0.95%	0.00	0.01	0.02	19.90%	7.96 MiB	19.90%	7.96 MiB	20.00 MiB	40.00 MiB	528.00 KiB	monitoring	0
prometheus-operator-5b7946f4d6-jp8wz	1.71%	0.00	0.10	0.20	13.51%	27.03 MiB	12.62%	25.23 MiB	100.00 MiB	200.00 MiB	104.00 KiB	monitoring	0
alertmanager-main-2	0.02%	0.00	0.10	0.10	24.33%	6.08 MiB	24.33%	6.08 MiB	25.00 MiB	25.00 MiB	16.00 KiB	monitoring	0
prometheus-k8s-0	0%	0	0.10	0.10	28.58%	7.14 MiB	28.58%	7.14 MiB	25.00 MiB	25.00 MiB	16.00 KiB	monitoring	0
cloud-rabbitmq-bb7686dfc-h8c22	1.03%	0.01	0.50	0.50	19.33%	98.98 MiB	18.48%	94.61 MiB	512.00 MiB	512.00 MiB	144.00 KiB	default	5
logstash-659f548d97-26n54	1.28%	0.01	0.50	0.50	60.58%	577.73 MiB	60.56%	577.58 MiB	953.67 MiB	953.67 MiB	15.06 MiB	default	0
node-exporter-hdlhd	1.34%	0.00	0.10	0.25	5.44%	9.79 MiB	5.44%	9.79 MiB	180.00 MiB	180.00 MiB	20.00 KiB	monitoring	0

咦，有红色。你看，有两个与 ES 相关的 POD，它们的 CPU 都飘红了，这下可有得玩了。既然是与 ES 相关的 POD，那我们就把 ES 所有的 POD 排个序看看。



elasticsearch-master-2	6.21%	0.12	2.00	2.00	66.71%	1.33 GiB	66.32%	1.33 GiB	2.00 GiB	2.00 GiB	22.80 MiB	default	0
elasticsearch-master-1	0.70%	0.01	2.00	2.00	70.17%	1.40 GiB	64.92%	1.30 GiB	2.00 GiB	2.00 GiB	61.27 MiB	default	0
elasticsearch-master-0	0.63%	0.01	2.00	2.00	70.69%	1.41 GiB	67.34%	1.35 GiB	2.00 GiB	2.00 GiB	142.69 MiB	default	0
elasticsearch-data-2	99.03%	0.99	1.00	1.00	93.29%	3.73 GiB	87.03%	3.48 GiB	4.00 GiB	4.00 GiB	12.51 MiB	default	0
elasticsearch-data-1	22.12%	0.22	1.00	1.00	93.00%	3.72 GiB	86.00%	3.44 GiB	4.00 GiB	4.00 GiB	10.31 MiB	default	0
elasticsearch-data-0	99.11%	0.99	1.00	1.00	93.40%	3.74 GiB	86.80%	3.47 GiB	4.00 GiB	4.00 GiB	50.23 MiB	default	0
elasticsearch-client-1	12.71%	0.13	1.00	1.00	85.11%	3.40 GiB	84.77%	3.39 GiB	4.00 GiB	4.00 GiB	3.34 MiB	default	0

从上图的数据来看，有一个 ES Client 消耗了 67% 的 CPU，有两个 ES Data 消耗了 99% 的 CPU，ES 本来就是吃 CPU 的大户，所以我们接下来要着重分析它。

这里我再说明一点，我们从前面的 worker 资源使用率一步一步走到这里，在分析方向上是合情合理的，因为这些都是属于我提到的全局监控的内容。

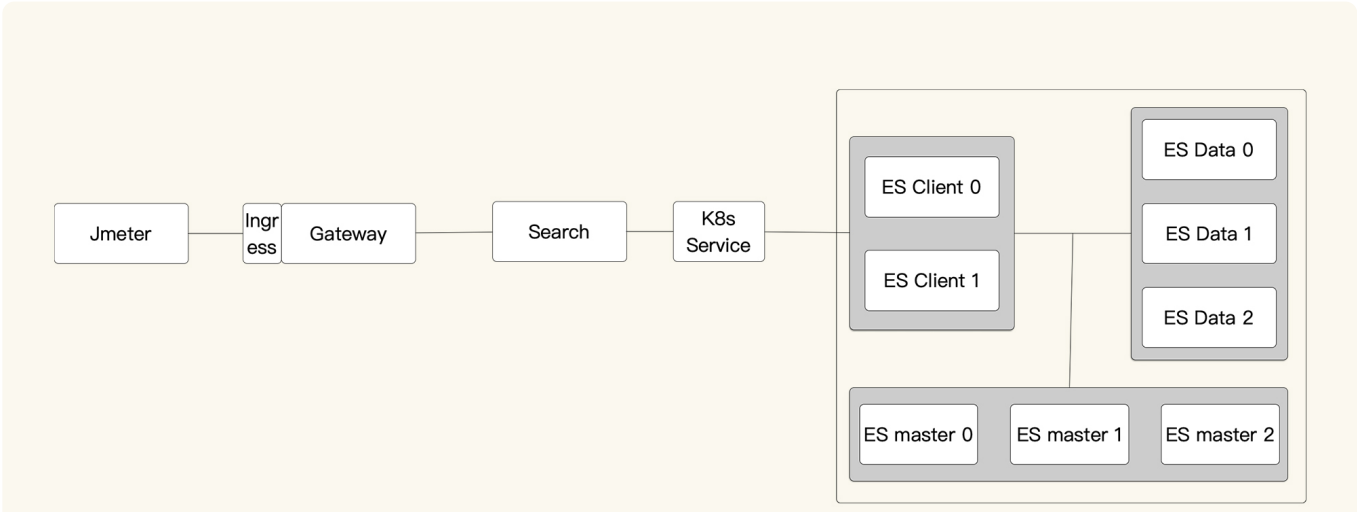
定向分析

现在我们就来扒一扒 ES，看看它在哪个 worker 节点上。罗列 Pod 信息如下：

```
[root@k8s-master-1 ~]# kubectl get pods -o wide | grep elasticsearch
elasticsearch-client-0                    1/1     Running   0          6h43m
elasticsearch-client-1                    1/1     Running   0          6h45m
elasticsearch-data-0                      1/1     Running   0          7h8m
elasticsearch-data-1                      1/1     Running   0          7h8m
elasticsearch-data-2                      1/1     Running   0          7h8m
elasticsearch-master-0                    1/1     Running   0          7h8m
elasticsearch-master-1                    1/1     Running   0          7h8m
elasticsearch-master-2                    1/1     Running   0          7h8m
[root@k8s-master-1 ~]#
```

现在就比较清晰了，可以看到，在整个 namespace 中有两个 ES client，三个 ES data，三个 ES master。

我们来画一个细一点的架构图，以便在脑子里记下这个逻辑：



再结合我们在全局分析中看到的资源使用率图，现在判断至少有两个问题：

1. ES client 请求不均衡；
2. ES data CPU 高。

下面我们一个一个来分析。

ES client 请求不均衡

从上面的架构图中可以看到，search 服务连两个 ES client，但是只有一个 ES client 的 CPU 使用率高。所以，我们需要查一下链路，看看 ES 的 service：

复制代码

```
[root@k8s-master-1 ~]# kubectl get svc -o wide | grep search
elasticsearch-client            NodePort    10.96.140.52    <none>        9200
elasticsearch-client-headless   ClusterIP   None            <none>        9200
elasticsearch-data              ClusterIP   10.96.16.151    <none>        9200
elasticsearch-data-headless     ClusterIP   None            <none>        9200
elasticsearch-master            ClusterIP   10.96.207.238   <none>        9200
elasticsearch-master-headless   ClusterIP   None            <none>        9200
svc-mall-search                 ClusterIP   10.96.27.150    <none>        8081
[root@k8s-master-1 ~]#
```

你看，整个 namespace 中有一个 client service（解析出来的是 VIP，访问此服务时不会绕过 K8s 的转发机制），还有一个 client-headless service（解析出来的是 POD IP，访问这个服务时会绕过 K8s 的转发机制）。

接下来，我们查一下为什么会出现访问不均衡的情况。

通过查看 search 服务的 ES 配置，我们看到如下信息：

复制代码

```
  elasticsearch:
    rest:
      uris: elasticsearch-client:9200
      username: elastic
      password: admin@123
```

看到我们这里是用的 elasticsearch-client:9200，我们再来看一下 client service 的配置：

复制代码

```
---
apiVersion: v1
kind: Service
metadata:
  annotations:
    meta.helm.sh/release-name: elasticsearch-client
    meta.helm.sh/release-namespace: default
  creationTimestamp: '2020-12-10T17:34:19Z'
  labels:
    app: elasticsearch-client
    app.kubernetes.io/managed-by: Helm
    chart: elasticsearch
    heritage: Helm
    release: elasticsearch-client
  managedFields:
    - apiVersion: v1
      fieldsType: FieldsV1
      fieldsV1:
        'f:metadata': {}
        'f:spec':
          'f:ports': {}
      manager: Go-http-client
      operation: Update
      time: '2020-12-10T17:34:19Z'
  name: elasticsearch-client
  namespace: default
  resourceVersion: '4803428'
  selfLink: /api/v1/namespaces/default/services/elasticsearch-client
  uid: 457e962e-bee0-49b7-9ec4-ebfbef0fecdd
spec:
  clusterIP: 10.96.140.52
  externalTrafficPolicy: Cluster
  ports:
    - name: http
      nodePort: 30200
      port: 9200
      protocol: TCP
      targetPort: 9200
    - name: transport
      nodePort: 31614
      port: 9300
      protocol: TCP
      targetPort: 9300
  selector:
    app: elasticsearch-client
    chart: elasticsearch
    heritage: Helm
```

```
48     release: elasticsearch-client
49   sessionAffinity: None
50   type: NodePort
```

从上面的配置来看，sessionAffinity 也配置为 None 了，也就是说这个 service 不以客户端的 IP 来保持 session。因为在这个环境配置中，Type 为 NodePort，而我们在 k8s 中配置的转发规则是 iptables。所以说，service 是依赖 iptables 的规则来做后端转发的。

接下来，我们检查一下 iptables 的转发规则。

我们先来看 iptables 中关于 ES client 的规则：

复制代码

```
1 [root@k8s-master-1 ~]# iptables -S KUBE-SERVICES -t nat|grep elasticsearch-cli
2 -A KUBE-SERVICES ! -s 10.100.0.0/16 -d 10.96.140.52/32 -p tcp -m comment --com
3 -A KUBE-SERVICES -d 10.96.140.52/32 -p tcp -m comment --comment "default/elast
4 [root@k8s-master-1 ~]#
```

可以看到，service 的规则名是 KUBE-SVC-XCX4XZ2WPAE7BUZ4，那我们再去查它对应的 iptables 规则：

复制代码

```
1 [root@k8s-master-1 ~]# iptables -S KUBE-SVC-XCX4XZ2WPAE7BUZ4 -t nat
2 -N KUBE-SVC-XCX4XZ2WPAE7BUZ4
3 -A KUBE-SVC-XCX4XZ2WPAE7BUZ4 -m comment --comment "default/elasticsearch-clien
4 [root@k8s-master-1 ~]#
5 [root@k8s-master-1 ~]# iptables -S KUBE-SEP-LO263M5QW4XA6E3Q -t nat
6 -N KUBE-SEP-LO263M5QW4XA6E3Q
7 -A KUBE-SEP-LO263M5QW4XA6E3Q -s 10.100.227.130/32 -m comment --comment "defaul
8 -A KUBE-SEP-LO263M5QW4XA6E3Q -p tcp -m comment --comment "default/elasticsearc
```

问题来了，这里好像没有负载均衡的配置（没有 probability 参数），并且根据 iptables 规则也只是转发到了一个 ES client 上。到这里，其实我们也就能理解，为什么在全局监控的时候，我们只看到一个 ES client 有那么高的 CPU 使用率，而另一个 ES client 却一点动静都没有。

但是，这里的 iptables 规则并不是自己来配置的，而是在部署 k8s 的时候自动刷进去的规则。现在只有一条规则了，所以只能转发到一个 POD 上去。

那我们就再刷一遍 ES 的 POD，重装一下 ES 的 POD，看 k8s 自己能不能刷出来负载均衡的 iptables 规则。重来一遍之后，我们再来看 iptables 规则：

复制代码

```
[root@k8s-master-1 ~]# iptables -S KUBE-SVC-XCX4XZ2WPAE7BUZ4 -t nat
-N KUBE-SVC-XCX4XZ2WPAE7BUZ4
-A KUBE-SVC-XCX4XZ2WPAE7BUZ4 -m comment --comment "default/elasticsearch-clien
-A KUBE-SVC-XCX4XZ2WPAE7BUZ4 -m comment --comment "default/elasticsearch-clien
[root@k8s-master-1 ~]#
```

哟，现在刷出来两条 iptables 规则了，看来之前在我们不断折腾的部署过程中，ES client 一直是有问题的。

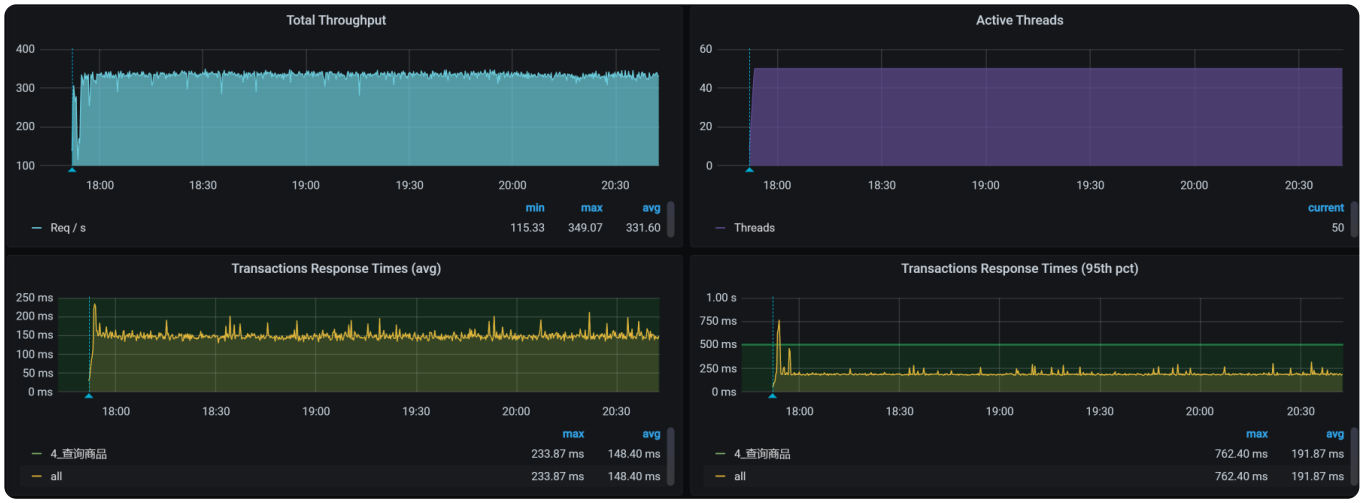
在上面的 iptables 规则里，那两条 iptables 的上一条中有一个关键词“——probability 0.50000000000”。我们知道，iptables 的匹配规则是从上到下的，既然上一条的匹配是随机 0.5，也就是说只有 50% 的请求会走第一条规则，那下一条自然也是随机 0.5 了，因为总共只有两条规则嘛。这样一来就均衡了。

我们再接着做这个接口的压力场景，看到如下信息：

Pod名称	CPU%(总核)	使用核数	CPU需求	CPU限制	WSS%	WSS	RSS%	RSS	内存需求	内存限制	磁盘	命名空间	重启
elasticsearch-client-0	20.32%	0.20	1.00	1.00	85.80%	3.43 GiB	83.86%	3.35 GiB	4.00 GiB	4.00 GiB	90.73 MiB	default	0
elasticsearch-client-1	20.04%	0.20	1.00	1.00	86.24%	3.45 GiB	84.15%	3.37 GiB	4.00 GiB	4.00 GiB	93.03 MiB	default	0
elasticsearch-data-0	17.76%	0.18	1.00	1.00	88.65%	3.55 GiB	84.89%	3.40 GiB	4.00 GiB	4.00 GiB	4.35 MiB	default	0
elasticsearch-data-1	91.11%	0.91	1.00	1.00	86.16%	3.45 GiB	85.77%	3.43 GiB	4.00 GiB	4.00 GiB	4.48 MiB	default	0
elasticsearch-data-2	19.48%	0.19	1.00	1.00	92.34%	3.69 GiB	84.69%	3.39 GiB	4.00 GiB	4.00 GiB	3.57 MiB	default	0
elasticsearch-master-0	0.34%	0.01	2.00	2.00	62.94%	1.26 GiB	62.66%	1.25 GiB	2.00 GiB	2.00 GiB	134.31 MiB	default	0
elasticsearch-master-1	0.84%	0.02	2.00	2.00	63.50%	1.27 GiB	63.23%	1.26 GiB	2.00 GiB	2.00 GiB	6.26 MiB	default	0
elasticsearch-master-2	0.78%	0.02	2.00	2.00	64.33%	1.29 GiB	64.07%	1.28 GiB	2.00 GiB	2.00 GiB	21.73 MiB	default	0

看起来 ES client 均衡了，对不对？

它对应的 TPS，如下：



明显 TPS 提升了 60 左右。

ES client 请求不均衡的问题解决了，现在，我们还要来看一下 ES data 单节点 CPU 高的问题。

ES Data CPU 使用率高

第一阶段：加一个 CPU

在 TPS 提升之后，我们再来看一下全局监控数据。

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.13 week	7.64 GiB	4	0.27	9.40%	23.99%	29.95%	0 B/s	76.97 kB/s	292	22	399.68 kbps	450.97 kbps
k8s-master-2	k8s-master-2	3.72 week	7.64 GiB	4	0.71	17.31%	23.19%	29.95%	0 B/s	74.56 kB/s	283	26	449.18 kbps	449.18 kbps
k8s-master-3	k8s-master-3	3.72 week	7.64 GiB	4	0.40	8.25%	24.43%	29.95%	0 B/s	75.46 kB/s	317	21	1.01 Mbps	1.01 Mbps
k8s-worker-1	k8s-worker-1	1.33 day	15.51 GiB	6	0.75	7.29%	52.41%	45.03%	9.21 kB/s	5.45 kB/s	29	91	10.06 Mbps	10.10 Mbps
k8s-worker-2	k8s-worker-2	1.33 day	15.51 GiB	6	1.58	23.39%	45.24%	29.95%	9.21 kB/s	129.66 kB/s	29	21 K	22.30 Mbps	26.99 Mbps
k8s-worker-3	k8s-worker-3	1.33 day	15.51 GiB	6	0.32	7.52%	39.26%	33.01%	13.81 kB/s	8.97 kB/s	30	245	177.39 kbps	223.91 kbps
k8s-worker-4	k8s-worker-4	2.63 week	15.51 GiB	6	0.47	11.91%	37.20%	31.58%	13.81 kB/s	739.54 kB/s	47	128	9.57 Mbps	10.50 Mbps
k8s-worker-5	k8s-worker-5	2.63 week	15.51 GiB	6	1.56	30.41%	57.74%	36.85%	18.41 kB/s	893.81 kB/s	28	113	10.98 Mbps	7.94 Mbps
k8s-worker-6	k8s-worker-6	2.63 week	15.51 GiB	6	0.84	16.47%	28.24%	39.06%	18.41 kB/s	7.17 kB/s	30	104	836.21 kbps	791.57 kbps
k8s-worker-7	k8s-worker-7	2.33 week	15.51 GiB	4	2.98	49.77%	46.22%	40.93%	13.81 kB/s	6.69 kB/s	29	328	16.23 Mbps	24.47 Mbps
k8s-worker-8	k8s-worker-8	1.15 week	15.51 GiB	4	0.37	9.35%	22.21%	44.28%	0 B/s	4.72 kB/s	28	222	136.04 kbps	107.80 kbps
k8s-worker-9	k8s-worker-9	2.33 week	15.51 GiB	6	0.50	12.83%	41.68%	29.95%	9.21 kB/s	15.50 kB/s	28	76	22.08 Mbps	31.56 Mbps

看起来比一开始好多了。基于前面分析 ES client 的经验，我们就先来查一下 ES data 的 iptables 规则：

复制代码

```
-- 查看下有哪些ES data的POD
[root@k8s-master-1 ~]# kubectl get pods -o wide | grep data
elasticsearch-data-0                    1/1     Running   0          10h
elasticsearch-data-1                    1/1     Running   0          10h
elasticsearch-data-2                    1/1     Running   0          10h

```

```
-- 查看ES data对应的iptables规则
[root@k8s-master-1 ~]# iptables -S KUBE-SERVICES -t nat|grep elasticsearch-dat
-A KUBE-SERVICES ! -s 10.100.0.0/16 -d 10.96.16.151/32 -p tcp -m comment --com
-A KUBE-SERVICES -d 10.96.16.151/32 -p tcp -m comment --comment "default/elast
-A KUBE-SERVICES ! -s 10.100.0.0/16 -d 10.96.16.151/32 -p tcp -m comment --com
-A KUBE-SERVICES -d 10.96.16.151/32 -p tcp -m comment --comment "default/elast


-- 查看9200（外部通信）对应的规则
[root@k8s-master-1 ~]# iptables -S KUBE-SVC-4LU6GV7CN63XJXEQ -t nat
-N KUBE-SVC-4LU6GV7CN63XJXEQ
-A KUBE-SVC-4LU6GV7CN63XJXEQ -m comment --comment "default/elasticsearch-data:
-A KUBE-SVC-4LU6GV7CN63XJXEQ -m comment --comment "default/elasticsearch-data:
-A KUBE-SVC-4LU6GV7CN63XJXEQ -m comment --comment "default/elasticsearch-data:


-- 查看以上三条规则的转发目标
[root@k8s-master-1 ~]# iptables -S KUBE-SEP-ZHLKOYKJY5GV3ZVN -t nat
-N KUBE-SEP-ZHLKOYKJY5GV3ZVN
-A KUBE-SEP-ZHLKOYKJY5GV3ZVN -s 10.100.18.197/32 -m comment --comment "default
-A KUBE-SEP-ZHLKOYKJY5GV3ZVN -p tcp -m comment --comment "default/elasticsearc
[root@k8s-master-1 ~]# iptables -S KUBE-SEP-6ILKZEZS3TMCB4VJ -t nat
-N KUBE-SEP-6ILKZEZS3TMCB4VJ
-A KUBE-SEP-6ILKZEZS3TMCB4VJ -s 10.100.251.67/32 -m comment --comment "default
-A KUBE-SEP-6ILKZEZS3TMCB4VJ -p tcp -m comment --comment "default/elasticsearc
[root@k8s-master-1 ~]# iptables -S KUBE-SEP-JOYLBDPA3LNXKWUK -t nat
-N KUBE-SEP-JOYLBDPA3LNXKWUK
-A KUBE-SEP-JOYLBDPA3LNXKWUK -s 10.100.5.5/32 -m comment --comment "default/el
-A KUBE-SEP-JOYLBDPA3LNXKWUK -p tcp -m comment --comment "default/elasticsearc
[root@k8s-master-1 ~]
```

Everythins is perfect!！规则很合理。ES Data 总共有三个 pod，从逻辑上来看，它们各占了三分之一。

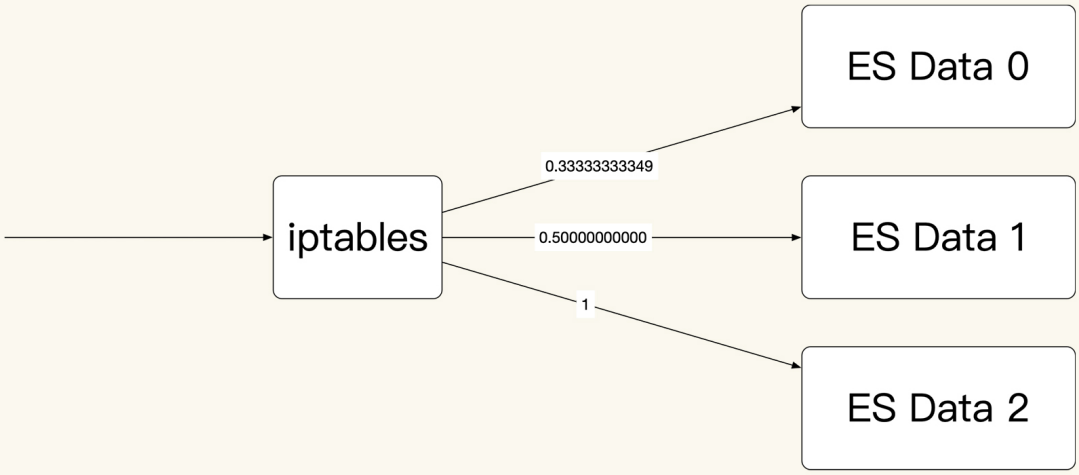
在前面的 ES client 分析中，我们讲到，第一个 POD 是 0.5，下一条自然也只剩下 0.5，这很容易理解。现在，ES data 的部分有三条 iptables 规则，我们来简单说明一下。

> 通常我们理解的 iptables 就是一个防火墙。不过，要是从根本上来讲，它不算是一个防火墙，只是一堆规则列表，而通过 iptables 设计的规则列表，请求会对应到 netfilter 框架中去，而这个 netfilter 框架，才是真正的防火墙。其中，netfilter 是处于内核中的，iptables 就只是一个用户空间上的配置工具而已。

> 我们知道 iptables 有四表五链。四表是：fileter 表（负责过滤）、nat 表（负责地址转换）、mangle 表（负责解析）、raw 表（关闭 nat 表上启用的连接追踪）；五链是：prerouting 链（路由前）、input 链（输入规则）、forward 链（转发规则）、output 链（输出规则）、postrouting 链（路由后）。

> 而在这一部分，我们主要是看 nat 表以及其上的链。对于其他的部分，如果你想学习，可自行查阅 iptables 相关知识。毕竟，我还时刻记得自己写的是一个性能专栏，而不是计算机基础知识专栏，哈哈。

从上面的信息可以看到，我们的这个集群中有三个 ES data 服务，对应着三条转发规则，其中，第一条规则的匹配比例是：0.33333333349；第二条比例：0.50000000000；第三条是 1。而这三条转发规则对应的 POD IP 和端口分别是：10.100.18.197:9200、10.100.251.67:9200、10.100.5.5:9200，这也就意味着，通过这三条 iptables 规则可以实现负载均衡，画图理解如下：



我们假设有 30 个请求进来，那 ES Data 0 上就会有 30x0.33333333349=10 个请求；对于剩下的 20 个请求，在 ES Data 1 上就会有 20x0.50000000000=10 个请求；而最后剩下的 10 个请求，自然就到了 ES Data 2 上。这是一个非常均衡的逻辑，只是在 iptables 规则中，我看着这几个数据比例，实在是觉得别扭。

既然明白了这个逻辑，下面我们还是把查询商品接口的场景压起来看一下：

Pod名称	CPU%(总核)	使用核数	CPU需求	CPU限制	WSS%	WSS	RSS%	RSS	内存需求	内存限制	磁盘	命名空间	重启
elasticsearch-client-0	20.99%	0.21	1.00	1.00	85.83%	3.43 GiB	83.88%	3.36 GiB	4.00 GiB	4.00 GiB	90.61 MiB	default	0
elasticsearch-client-1	20.34%	0.20	1.00	1.00	86.26%	3.45 GiB	84.16%	3.37 GiB	4.00 GiB	4.00 GiB	91.84 MiB	default	0
elasticsearch-data-0	20.51%	0.21	1.00	1.00	92.43%	3.70 GiB	84.90%	3.40 GiB	4.00 GiB	4.00 GiB	6.29 MiB	default	0
elasticsearch-data-1	99.84%	1.00	1.00	1.00	86.23%	3.45 GiB	85.80%	3.43 GiB	4.00 GiB	4.00 GiB	4.48 MiB	default	0
elasticsearch-data-2	22.03%	0.22	1.00	1.00	92.35%	3.69 GiB	84.72%	3.39 GiB	4.00 GiB	4.00 GiB	4.89 MiB	default	0
elasticsearch-master-0	0.38%	0.01	2.00	2.00	62.98%	1.26 GiB	62.67%	1.25 GiB	2.00 GiB	2.00 GiB	94.31 MiB	default	0
elasticsearch-master-1	0.85%	0.02	2.00	2.00	63.54%	1.27 GiB	63.24%	1.26 GiB	2.00 GiB	2.00 GiB	6.14 MiB	default	0
elasticsearch-master-2	0.69%	0.01	2.00	2.00	64.38%	1.29 GiB	64.08%	1.28 GiB	2.00 GiB	2.00 GiB	22.73 MiB	default	0

从数据上来看，经常会出现 ES data 某个节点消耗 CPU 高的情况。可是，对应到我们前面看到的全局 worker 监控界面中，并没有哪个 worker 的 CPU 很高。所以，在这里，我们要查一下 ES Data 中的 cgroup 配置，看它的限制是多少。

也就是说，ES data 的每个 POD 都是配置了一颗 CPU，难怪 CPU 使用率动不动就红了。

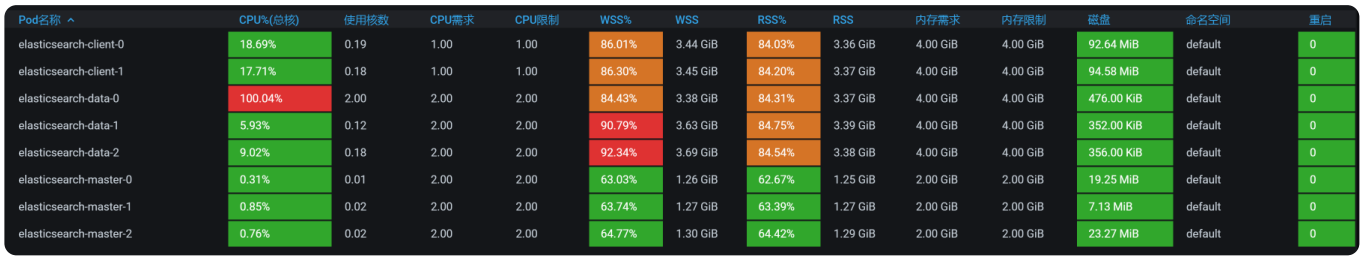
还有一点你要记住，前面我们在查看 data 列表的时候发现，ES data 0 在 worker-5 上，ES data 1 在 worker-7 上，ES data 2 在 worker-9 上。而我们现在看到的却是，它们都各自分到了一个 CPU。既然如此，那我们就再添加一个 CPU，然后再回去看一下 worker-5/7/9 的反应。为什么只加一个 CPU 呢？因为从 worker-7 上来看，现在的 CPU 使用率已经在 50% 左右了，要是加多了，我怕它吃不消。

看一下压力场景执行的效果：

似乎……不怎么样？TPS 并没有增加。

第二阶段：加副本

我们再看加了 CPU 之后的全局 POD 监控：



Pod名称	CPU%(总核)	使用核数	CPU需求	CPU限制	WSS%	WSS	RSS%	RSS	内存需求	内存限制	磁盘	命名空间	重启
elasticsearch-client-0	18.69%	0.19	1.00	1.00	86.01%	3.44 GiB	84.03%	3.36 GiB	4.00 GiB	4.00 GiB	92.64 MiB	default	0
elasticsearch-client-1	17.71%	0.18	1.00	1.00	86.30%	3.45 GiB	84.20%	3.37 GiB	4.00 GiB	4.00 GiB	94.58 MiB	default	0
elasticsearch-data-0	100.04%	2.00	2.00	2.00	84.43%	3.38 GiB	84.31%	3.37 GiB	4.00 GiB	4.00 GiB	476.00 KiB	default	0
elasticsearch-data-1	5.93%	0.12	2.00	2.00	90.79%	3.63 GiB	84.75%	3.39 GiB	4.00 GiB	4.00 GiB	352.00 KiB	default	0
elasticsearch-data-2	9.02%	0.18	2.00	2.00	92.34%	3.69 GiB	84.54%	3.38 GiB	4.00 GiB	4.00 GiB	356.00 KiB	default	0
elasticsearch-master-0	0.31%	0.01	2.00	2.00	63.03%	1.26 GiB	62.67%	1.25 GiB	2.00 GiB	2.00 GiB	19.25 MiB	default	0
elasticsearch-master-1	0.85%	0.02	2.00	2.00	63.74%	1.27 GiB	63.39%	1.27 GiB	2.00 GiB	2.00 GiB	7.13 MiB	default	0
elasticsearch-master-2	0.76%	0.02	2.00	2.00	64.77%	1.30 GiB	64.42%	1.29 GiB	2.00 GiB	2.00 GiB	23.27 MiB	default	0

还是只有一个 ES data 的 CPU 使用率高，所以我想查一下 ES 中的数据分布。因为负载均衡的问题解决了，并且知道有三个 ES data 节点。现在我们就要知道是不是每个节点都被访问到了。

```
复制代码
1  pms                0 p 10.100.18.199 _w   32   17690 18363   6.
2  pms                0 p 10.100.18.199 _15  41    2110     0 465.
3  pms                0 p 10.100.18.199 _16  42   21083 30255   9.
4  pms                0 p 10.100.18.199 _17  43    2572     0   56
5  pms                0 p 10.100.18.199 _18  44    1403     0 322.
6  pms                0 p 10.100.18.199 _19  45    1856     0 414.
7  pms                0 p 10.100.18.199 _1a  46    1904     0   42
```

为啥数据都在一个节点上（都是 10.100.18.199）？看起来只有一个数据副本的原因了。

```
green open pms                                   A--6O32bQaSBrJJPJJwQL
```

复制代码

所以，我们先把副本数加上去，因为我们有三个 data 节点，所以这里加三个副本：

复制代码

```
PUT /pms/_settings
{
    "number_of_replicas": 3
}

我们再次查看ES中的数据分布，如下所示：

pms                          0 r 10.100.18.200 _w   32   17690 18363   6.
pms                          0 r 10.100.18.200 _15  41    2110      0 465.
pms                          0 r 10.100.18.200 _16  42   21083 30255   9.
pms                          0 r 10.100.18.200 _17  43    2572      0   56
pms                          0 r 10.100.18.200 _18  44    1403      0 322.
pms                          0 r 10.100.18.200 _19  45    1856      0 414.
pms                          0 r 10.100.18.200 _1a  46    1904      0   42
pms                          0 p 10.100.251.69 _w   32   17690 18363   6.
pms                          0 p 10.100.251.69 _15  41    2110      0 465.
pms                          0 p 10.100.251.69 _16  42   21083 30255   9.
pms                          0 p 10.100.251.69 _17  43    2572      0   56
pms                          0 p 10.100.251.69 _18  44    1403      0 322.
pms                          0 p 10.100.251.69 _19  45    1856      0 414.
pms                          0 p 10.100.251.69 _1a  46    1904      0   42
pms                          0 r 10.100.140.10 _w   32   17690 18363   6.
pms                          0 r 10.100.140.10 _15  41    2110      0 465.
pms                          0 r 10.100.140.10 _16  42   21083 30255   9.
pms                          0 r 10.100.140.10 _17  43    2572      0   56
pms                          0 r 10.100.140.10 _18  44    1403      0 322.
pms                          0 r 10.100.140.10 _19  45    1856      0 414.
pms                          0 r 10.100.140.10 _1a  46    1904      0   42
```

我们接着压起来，看看 POD 的资源：

Pod名称	CPU%(总核)	使用核数	CPU需求	CPU限制	WSS%	WSS	RSS%	RSS	内存需求	内存限制	磁盘	命名空间	重启
elasticsearch-client-0	49.63%	0.50	1.00	1.00	86.03%	3.44 GiB	84.05%	3.36 GiB	4.00 GiB	4.00 GiB	91.83 MiB	default	0
elasticsearch-client-1	46.48%	0.46	1.00	1.00	86.31%	3.45 GiB	84.20%	3.37 GiB	4.00 GiB	4.00 GiB	93.43 MiB	default	0
elasticsearch-data-0	99.75%	2.00	2.00	2.00	91.99%	3.68 GiB	84.54%	3.38 GiB	4.00 GiB	4.00 GiB	476.00 KiB	default	0
elasticsearch-data-1	91.60%	1.83	2.00	2.00	93.38%	3.74 GiB	84.85%	3.39 GiB	4.00 GiB	4.00 GiB	608.00 KiB	default	0
elasticsearch-data-2	82.51%	1.65	2.00	2.00	87.66%	3.51 GiB	84.82%	3.39 GiB	4.00 GiB	4.00 GiB	612.00 KiB	default	0
elasticsearch-master-0	0.36%	0.01	2.00	2.00	63.07%	1.26 GiB	62.70%	1.25 GiB	2.00 GiB	2.00 GiB	19.25 MiB	default	0
elasticsearch-master-1	0.87%	0.02	2.00	2.00	63.79%	1.28 GiB	63.44%	1.27 GiB	2.00 GiB	2.00 GiB	15.07 MiB	default	0
elasticsearch-master-2	0.90%	0.02	2.00	2.00	64.89%	1.30 GiB	64.54%	1.29 GiB	2.00 GiB	2.00 GiB	56.25 MiB	default	0

现在看着是不是开心多了？data 节点的 CPU 都用起来了。

我们再看一下 worker 的资源：

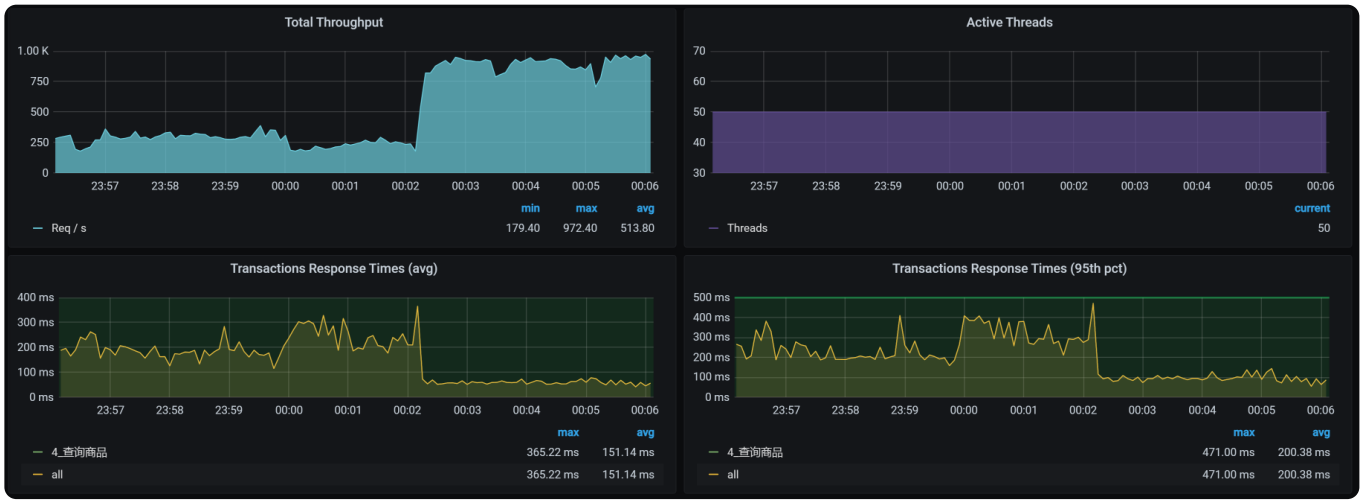
```
[root@k8s-master-1 ~]# kubectl get pods -o wide | grep data
elasticsearch-data-0                    1/1     Running   0          16m
elasticsearch-data-1                    1/1     Running   0          17m
elasticsearch-data-2                    1/1     Running   0          18m
```

现在 ES Data 的 POD 分布到 2、5、9 三这个 worker 上去了，我们查看下全局监控：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.15 week	7.64 GiB	4	0.39	9.82%	24.01%	29.95%	0 B/s	74.53 kB/s	293	24	406.69 kbps	427.58 kbps
k8s-master-2	k8s-master-2	3.73 week	7.64 GiB	4	0.76	18.14%	22.98%	29.95%	0 B/s	70.95 kB/s	282	26	249.71 kbps	261.15 kbps
k8s-master-3	k8s-master-3	3.74 week	7.64 GiB	4	0.23	8.89%	24.46%	29.95%	0 B/s	75.61 kB/s	318	24	991.91 kbps	991.91 kbps
k8s-worker-1	k8s-worker-1	1.44 day	15.51 GiB	6	1.75	13.05%	52.13%	44.92%	9.21 kB/s	6.62 kB/s	29	85	20.00 Mbps	19.79 Mbps
k8s-worker-2	k8s-worker-2	1.44 day	15.51 GiB	6	8.70	74.13%	67.40%	29.95%	13.81 kB/s	338.21 kB/s	29	25 K	54.82 Mbps	78.78 Mbps
k8s-worker-3	k8s-worker-3	1.44 day	15.51 GiB	6	0.31	7.52%	39.53%	33.01%	13.81 kB/s	9.58 kB/s	30	252	186.98 kbps	251.57 kbps
k8s-worker-4	k8s-worker-4	2.65 week	15.51 GiB	6	1.93	12.65%	37.09%	31.78%	9.21 kB/s	938.41 kB/s	47	111	13.71 Mbps	14.64 Mbps
k8s-worker-5	k8s-worker-5	2.64 week	15.51 GiB	6	7.43	58.15%	57.65%	36.54%	13.81 kB/s	775.42 kB/s	28	116	18.09 Mbps	43.42 Mbps
k8s-worker-6	k8s-worker-6	2.64 week	15.51 GiB	6	5.13	16.13%	28.52%	39.06%	27.62 kB/s	7.68 kB/s	30	114	951.19 kbps	905.85 kbps
k8s-worker-7	k8s-worker-7	2.34 week	15.51 GiB	4	2.86	57.09%	24.07%	40.92%	18.42 kB/s	6.00 kB/s	29	693	40.24 Mbps	40.18 Mbps
k8s-worker-8	k8s-worker-8	1.17 week	15.51 GiB	4	0.52	12.11%	22.26%	44.28%	0 B/s	3.54 kB/s	28	306	2.23 Mbps	377.80 kbps
k8s-worker-9	k8s-worker-9	2.34 week	15.51 GiB	6	1.75	34.94%	41.56%	29.95%	13.81 kB/s	4.53 kB/s	28	82	14.92 Mbps	31.80 Mbps

嗯，不错，ES data 的 POD 把资源用起来了。其实这里要想继续调，还可以把 CPU 加大，ES 本来就是吃 CPU、内存的大户。不过，我们前面在配置的时候，给 ES data 的 CPU 也确实太小了。这个问题，并不是我故意设计出来的，而是当时在部署的时候，没考虑到这些。

最后，我们来看优化后的效果：



呀呀呀，你看 TPS 压都压不住呀，很快涨到 900 左右了！这个优化结果很好。

现在回过头来看第一个阶段，我们加 CPU 没有效果，主要还是因为副本数量太少。其实，在 ES 的优化中，还有很多细节可以玩。只不过，在我们这个课程中，我希望给你的是一个整体的分析思路和逻辑，而不是纠结于每个细节上的参数。所以，在这里，我们就不再说具体参数的调整了。

如果你想在 ES 上做更多的优化，可以在分析完业务之后，确定一下 ES 的架构、数据索引、分片等信息，然后再来设计一个合理的 ES 部署。

总结

在这节课中，我们看到 APM 工具也有无能为力的地方。所以说，当我们分析到一个具体组件之后，要想再往下分析，就得靠自己的技术功底了。

在出现请求不均衡的时候，我们一定要先去看负载均衡的逻辑有没有问题。当看到 ES client 不均衡时，我们去看了 iptables 的原理，在发现 iptables 只有一个转发规则的时候，接下来要做的当然就是重刷转发规则了。

在 ES client 转发均衡了之后，我们在 ES data 单节点上又看到 CPU 使用率过高。由于 ES data 在 POD 当中，我们自然就要想到去看 cgroup 的限制。

而在添加了 CPU 之后，我们发现 TPS 并没有提高，这时候就要去看 ES 的逻辑了。ES 的强大之处就在于多副本多分片的查询能力，所以，我们增加了副本之后，CPU 就用起来了，这是一个合理的优化结果，TPS 也自然提高了。

经过一系列的动作，我们终于把资源给用起来了。这也是我一直在强调的，**性能优化第一个阶段的目标，就是把资源给用起来，然后再考虑更细节的优化。**

课后作业

最后，我给你留两道题，请你思考一下：

1. 当负载出现不均衡时，主要的分析方向是什么？
2. 什么时候才需要去看组件内部的实现逻辑？

记得在留言区和我讨论、交流你的想法，每一次思考都会让你更进一步。

如果你读完这篇文章有所收获，也欢迎你分享给你的朋友，共同学习进步。我们下一讲再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 14 | 用户信息查询：如何解决网络软中断瓶颈问题？

下一篇 16 | 商品加入购物车：SQL优化和压力工具中的参数分析

精选留言 (2)

写留言

道长
2021-04-28

今天这篇，感觉有点难，可能是没接触过k8s吧
还有负载不均衡我肯定是先去看负载策略，再往下排查；只有当策略都没没问题我才会考虑内部逻辑，个人看法

展开

1

继贤
2021-04-23

老师，你讲的性能调优要先将资源利用起来，这个资源使用到达多少就算是合理了，比如CPU，有些系统是80%，有些可以到100，该怎么衡量？

展开

作者回复: 要分析下资源是用到了哪里才能判断合理不合理。在性能测试中，可以用到100%，但是在生产上是不可以的。所以资源使用到多少要看场景和业务影响。